JavaScript Numeric Bugs Field Guide

Floating-point pitfalls, conversion functions, and the exact test cases that catch numeric defects before production.

1. parseInt vs. parseFloat vs. Number()

Function	Input	Result	Behavior
`parseInt("42px")`	"42px"	42	Reads digits until non-numeric, then stops
`parseFloat("42.5px")`	"42.5px"	42.5	Same partial read, keeps decimal
`Number("42px")`	"42px"	NaN	Requires the entire string to be numeric
`Number("42")`	"42"	42	Clean numeric strings convert fine on all three

2. Core Math Object Methods

Method	Behavior	Watch For
`Math.round()`	Rounds .5 up, always	Math.round(-2.5) returns -2, not -3
`Math.floor()` / `ceil()`	Always rounds one direction	Off-by-one page counts often trace here
`Math.max()` / `min()`	Largest/smallest argument	Returns NaN if any argument isn't numeric
`Math.random()`	Float from 0 up to (not incl.) 1	Not cryptographically secure — flag near auth/tokens

3. Test Cases That Catch Numeric Defects

Currency values known to trigger rounding error
Test 0.1 + 0.2 style additions across multiple line items. If the result isn't exact, currency math isn't using integer cents.

Malformed numeric strings
Submit "5mg", "42px", or values with trailing units into numeric fields. parseInt/parseFloat will silently accept them.

NaN comparison logic
Confirm invalid-number checks use Number.isNaN(value), not === NaN, which always evaluates false.

Large numeric IDs
Check IDs near or beyond 2^53 (9,007,199,254,740,991) — precision degrades silently past this point.

Loose vs. strict equality
Look for == instead of === in numeric assertions; type coercion can mask a string-vs-number bug.

Fix pattern for currency math: convert dollar amounts to integer cents before any addition or multiplication, then divide back to dollars only for display. This removes compounding floating-point error at the source instead of masking it with toFixed().